Functional Programming

- every function must **take an argument** (or more than one)
- every function must return a value (`void` return forbidden)
- functions are **pure**: they should have no side effects
  - side effects include changing global variables, input/output, print statements, etc.
- calling a function `f(x)` with the same argument `x` should **always return the same output `y`**
- variables are **immutable**: after declaration, they cannot be modified
- functions are **first-class** - just like other values, they can be stored as variables, passed as arguments, etc.

Formally, a **pure function** (referentially transparent function) has two core properties:

1. It doesn't depend on any data other than its inputs to compute a result
2. It doesn't modify any data beyond initializing local variables required to compute its output

consequences:

- no random numbers can be generated or used (violate determinism)
- no input is taken in a compiled Haskell program
- no output is produced (aside from the final product) in a compiled Haskell program
  - since that would not longer be pure (we are altering program state)

Functional versus Imperative

Imperative programs:

- are sequences of statements, loops, and function calls - they operate sequentially
- require changes in variable state
- as a result, multi-threading can be tricky, since there's shared mutable state and thus race conditions
- the order of execution matters! because of side-effects

Functional programs have different paradigms:

- at the core, functional programs are **only composition of functions** - no loops, statements, etc.
- no changes in variable state are allowed
- multi-threading is trivial: there is no shared mutable state (since mutability is not allowed)!
- and, interestingly, **the order of execution is not important**
  - because no side-effects

Intro to Haskell

Indentation

- Code that's part of an expression should be indented further in than the beginning of that expression.
- You must align the spacing for all items in a group.

Data Types in Haskell

The Type System

- **statically typed language** - the type of *all* variables and *all functions* can be figured out at **compile time**
- has **type inference** - the compiler can figure out the types of most expressions *without you annotating them*
  - this started in functional programming, and has made its way into mainstream languages (`auto` in C++)

Primitive Types

- `Int`: 64-bit signed integers
- `Integer`: arbitrary-precision signed integers
- `Bool`: booleans (`True` or `False`)
- `Char`: characters (like in other languages)
- `Float`: 32-bit (single-precision) floating point
- `Double`: 64-bit (double-precision) floating point
- `:t` tells us the type of the expression passed to it

Operators

- the `/` division and `mod` operator **does not work on Integers**!
  - use `` `div` ``, `` `mod` ``
- To make numbers negative, you need to add a parentheses: this makes `-` a *unary operator*: `(-)1`
- use prefix notation for infix operators by wrapping in parenthesis: `(+) 1 2`
- use infix notation for prefix operators by wrapping in backticks: `2 `elem` [2,3,4]`

Composite Types

- a **`Tuple`** is a **fixed-size** collection of values; each value can be any type.
- a **`List`** is a sequence of values; **each value must be the same type**.
- a **`String`** is a list of characters!

Tuples

Tuples are great for associating values together, or having a function "return multiple values".

- `fst` returns first element
- `snd` returns second element
  - these are only defined for tuples with two elements

The **type of the tuple** encodes:

- the number of elements
- the types of each element

so you can't have a list of any two tuples - they must all be tuples of the same size and types!

```
grade = ("Sergey", 97)
whoa = ((1,"two"),['a','b','c'], 17)
```

```
-- ghci> :t grade
-- grade :: (String, Integer)
```

Lists

- **not arrays**: they have `O(N)` access and no pre-defined size.

```
primes :: [Int]
primes = [1,2,3,5,7,11,13]
```

- `head` returns the first item of the list (not in a list)
- `tail` returns a list of the rest of the items in a list
- `null` returns True if a list is empty
- `length` returns its length
- `take` and `drop` are convenient ways to access slices of a list.
- `!!` gives random access by index (0-indexed)
- `elem` returns True if an item is in a list:

```
ghci> take 3 primes
[1,2,3]
ghci> drop 4 primes
[7,11,13]
ghci> primes !! 2
3
ghci> elem 2 primes
True
```

- `sum` adds up the entire list; `prod` returns product of list
- `or` performs boolean `or` over the list; there's also a corresponding `and`
- `zip` creates tuples out of two lists (same as Python)

```
ghci> zip [10,20,30] ["SWE", "Chef", "Writer"]
[(10,"SWE"),(20,"Chef"),(30,"Writer")]
```

Strings

- a string is just list of chars--can use all list functions
- perform concatenation with `++`:

List Processing

Concatenate and Cons

- `:` "cons" operator adds item to *front* of list
- `++` concatenates two lists
- can create lists from scratch - but need the trailing `: []`

```
ghci> 1 : 3 : []
[1,3]
```

Ranges

- create a list by specifying a range (Inclusive!)

```
one_to_ten = [1..10]          -- All #s between 1 and 10, inclusive
oddities = [1,3..10]          -- Odd #s between 1 and 10
whole_lotta_numbers = [42..] -- infinite list from 42 onward!
tricycle = cycle [1,3,5]      -- infinite cycle of 1,3,5,1,3,5,…
```

- infinite lists are possible because Haskell is **lazily-evaluated**
- won't generate list items from a range *until they're needed*
- **none of the list is initialized when it's initially declared**
- can you take the `tail` of an infinite list?
    - Yes! `tail [42..]` is the same as `[43..]`
    - `tail [42..] == [43..]` hangs, and `tail [42..]` just keeps on printing numbers. What's up with that?
    - when `ghci` returns a value, it calls a "print function" on it; the print function for lists keeps on printing until the list is empty.
    - `tail [42..] == [43..]` hangs because of the **implementation** of the `==` operator, which compares heads until the list is empty. This never happens so program never terminates

List Comprehensions

a compact yet powerful syntax to create lists

Semiformally:

> *A list comprehension is a F.P. construct that lets you easily create a new list based on one or more existing lists. With a list comprehension, you specify the following inputs:*
>
> 1. *One or more input lists that you want to draw from (called "**generators**")*
> 2. *A set of filters on the input lists (these are called "**guards**")*
> 3. *A **transformation** applied to the inputs before items are added to the output list.*

Syntax:

```
[f x y... | x <- list1, y <- list2, ..., guardA x, guardB y ]
```

```
# equivalent to this
output = []
for x in list1:
  for y in list 2:
    if guardA(x) and guardB(y):
      output.append(f(x,y))
```

- similar to set builder notation
- generators (list1, list2) create values of x and y
    - iterates like nests for loops:
- guards, at the end, filter out bad values
- can then apply some arbitrary function to the filtered values

Examples:

```
squares = [x^2 | x <- [2..7]]
-- [4,9,16,25,36,49]
```

```haskell
prods = [x*y | x <- [3,7], y <- [2,4,6]]
-- [6,12,18,14,28,42]

-- Generate combinations of tuples
menu = [(oz,flavor) |
   oz <- [4,6],
  flavor <- ["Choc","Earwax","Booger"]]
-- [(4,"Choc"),(4,"Earwax"),(4,"Booger"),(6,"Choc"),(6,"Earwax"),(6,"Booger")]
```

Functions

Three core components:

1. (optional): type information about the function's **parameters** and **return value**
2. the **function name** and **parameter** names
3. an **expression** that defines the function's behaviour

```haskell
insult :: String -> String -> String
insult name smell =
  name ++ " smells like " ++ smell ++ " and doesn't floss."
```

- there's no explicit `return` since the right-hand side of a function *is* what it returns!
- spaces and indentation are used to define code blocks; no braces necessary

Optional Type Declarations

Type declaration is optional because of Haskell's type inference, but it's best practice

- type declarations can provide better compiler warnings and errors
- type declarations make your code **easier to read**
- and, it can help you think about how to write the function itself!

Main Functions and (avoiding) Monads

How can you have a main function, and how can you get user input which is a side effect?

To do it, we'll create a function called `main`, and annotate it with the `IO()` "type" (really - a system and monad). We then add this strange `do` block, and write what seems like imperative code:

```haskell
-- insult.hs
insult :: String -> String -> String
insult name smell =
  name ++ " smells like " ++ smell ++ " and doesn't floss."

main::IO()
main = do
  putStr "What is your name? "
  name <- getLine
  putStrLn (insult name "unwashed dog")
```

Examples

```haskell
isBigger :: Double -> Double -> Bool
isBigger a b = a > b

average :: [Double] -> Double
average list =
  (sum list) / fromIntegral (length list)
```

`fromIntegral` is used to cast the value of `length` to be divisible, and said it maps from `Integral -> Double`

```haskell
get_last_item :: [any_type] -> any_type
get_last_item lst =
  head (reverse lst)
```

- This function works for a list of any type
- the `any_type` is a **type variable**, which we can think of as a generic or template type
- type variables *must* be defined with a lowercase first letter

Precedence

```haskell
f :: Int -> Int
f x = x^2

g :: Int -> Int
g x = 3 * x

y = f g 2
z = f 5 * 10
```

- `y = f g 2` is a **syntax error**!
  - Haskell evaluates functions **from left to right** (keyword: **left-associative**)
  - so, `f g 2` is the same as `((f g) 2)`, which doesn't make sense - since `f` doesn't operate on functions
- `f 5 * 10 = 250`
  - functions have higher precedence than operators, so they're always evaluated first
  - in other words, this is `(f 5) * 10`

```haskell
-- f(g(x))
compute_f_of_g x = f (g x)

-- f(x * 10)
compute_f_of_x_times_ten x = f (x * 10)
```

Local Bindings

The `let` keyword

```
-- let.hs
get_nerd_status gpa study_hrs =
  let
      gpa_part = 1 / (4.01 - gpa)
      study_part = study_hrs * 10
      nerd_score = gpa_part + study_part
  in
      if nerd_score > 100 then
           "You are super nerdy!"
      else "You're a nerd poser."
```

The `let` construct consists of two parts:

- the first part is between the `let` and the `in`; here, you define one or more "bindings" to associate a name with an expression
  - for example, the third line of the function binds the name `gpa_part` to the expression `1 / (4.01 - gpa)`.
- the second part follows `in`; it contains an expression where the bindings are used.

The bindings are immutable

The `where` keyword

The `where` keyword is similar. First write the code, then specify the bindings after the `where` keyword

```
-- let.hs
get_nerd_status gpa study_hrs =
  if nerd_score > 100
     then "You are super nerdy!"
     else "You're a nerd poser."
  where
      gpa_part = 1 / (4.01 - gpa)
      study_part = study_hrs * 10
      nerd_score = gpa_part + study_part
```

When would you use `let` and when do you use `where`? Generally

- when defining bindings (variables) for *a single expression* (e.g. the example here), either one is fine
- when defining bindings for use *across multiple expressions*, use `where`

Nested functions with `let` or `where`

- can also use `let` and `where` to define *nested functions*:
- notice that nested functions can use all of their enclosing function's bindings

```
-- nestfunc.hs
-- Function to describe someone's behavior
whats_the_behavior_of name =
 if name == "Carey"
    then behaves_like "twelve year-old"
    else behaves_like "grown-up"
```

```
  where
    behaves_like what =
      name ++ " behaves like a " ++ what ++ "!"
```

Lazy execution with `let` and `where`

```
potentially_slow_func arg =
  let
      val1 = really_slow_function arg
      val2 = very_fast_function arg
      val3 = pretty_fast_function arg
  in
      if val3 > 100 then val1
                    else val2
```

Haskell's behavior when running the code above is:

- in the `let` block, Haskell binds `val1`, `val2`, and `val3` to the expressions - but *doesn't* evaluate them!
- when the function body runs, the `if` condition requires `val3`.
- so, Haskell evaluates the expression associated with `val3`, calling `pretty_fast_function`.
- then, *only one* of `val1` and `val2` will evaluate.
- so, if `pretty_fast_function` returns 20, then we go to the `else` branch. To compute `val2` `very_fast_function` is called. We will *never* call `really_slow_function` in this case!

Note that Haskell doesn't have to re-evaluate the binding every time it's used - after doing it once, it can just cache it!

Control Flow

`if` - `then` - `else`

```
if <expression> then <expression>
                else <expression>
```

every `if` statement must have an `else`. Otherwise our function might not return anything

Guards

```
  | <condition> = <expr>  -- if <condition> then <expr>
```

We can define a function as a series of one or more guards, like

```
somefunc param1 param2
  | <if-x-is-true> = <run-this>
  | <if-y-is-true> = <run-that>
  | <if-z-is-true> = <run-the-other>
  | otherwise = <run-this-otherwise>
```

- no equal sign after defining the function

- each guard begins with `|`
- after that, the guard contains two parts:
  - the first part is a *boolean expression*; we want to evaluate to see if it is `True`.
  - the second part is the "payload" expression. It will return if the expression is `True`.
- Haskell evaluates all of the guards **from top to bottom**, and returns the payload of the first guard that is `True`.
- `otherwise` acts like a wildcard (always evaluates to true)

quicksort:

```
qsort lst
 | lst == [] = []
 | otherwise = less_eq ++ [pivot] ++ greater
 where
    pivot = head lst
    rest_lst = tail lst
    less_eq = qsort [a | a <- rest_lst, a <= pivot]
    greater = qsort [a | a <- rest_lst, a > pivot]
```

Pattern Matching

First, we'll explore parameter-based pattern matching, where:

- we define multiple versions of the same function
- each version of the function must have the same number and types of arguments
- when the function is called, Haskell checks each definition in-order from top to bottom, and calls the *first* function which has matching parameters

Pattern Matching with Constants

- we can place constants instead of the name of an expected argument.
- if multiple patterns match, first match is used
- works with other types, like numbers or lists

```
genz_critic :: String -> String -> String
genz_critic "Carey" word = "Oh Carey... OK boomer!"
genz_critic name "lit" = name ++ ", you're a lit genz-er"
genz_critic name word = name ++ ", " ++ word ++ " is not gucci!"


list_len [] = 0
list_len lst = 1 + list_len (tail lst)
```

Pattern Matching with Tuples

- can match each item in tuple

```
-- Original way
exp :: (Int,Int) -> Int
exp t = (fst t) ^ (snd t)

-- Version #2: With pattern matching
```

```haskell
expv2 :: (Int,Int) -> Int
expv2 (a,b) = a ^ b

-- Version #3: Adding a constant parameter
expv3 :: (Int,Int) -> Int
expv3 (a,0) = 1
expv3 (a,b) = a ^ b
```

- extract values out of tuples
- `_` is a wildcard: anything can match this item
  - can use multiple `_` without clashing

```haskell
extract_1st :: (type1,type2,type3) -> type1
extract_1st (a,_,_) = a

extract_2nd :: (type1,type2,type3) -> type2
extract_2nd (_,b,_) = b

extract_3rd :: (type1,type2,type3) -> type3
extract_3rd (_,_,c) = c
```

New learners are often tempted to write code like:

```haskell
tuple_eq (a, a) = True   -- WRONG!!!
tuple_eq _ = False

-- using guards
tuple_eq (a, b)
  | a == b = True
  | otherwise = False

-- or, use (==) directly
tuple_eq (a, b) = a == b
```

Haskell will complain about "conflicting definitions for `'a'`" on the first line. To compare the two values obtained from pattern matching, they have to be bound to different names and compared afterwards.

Pattern Matching with Lists

```haskell
-- Extract the first item from a list
get_first :: [a] -> a
get_first (x:xs) = x

-- Extract all but the first item of a list
get_rest :: [a] -> [a]
get_rest (x:xs) = xs

-- Extract the second item from a list
get_second :: [a] -> a
get_second (x:y:xs) = y
```

- can think of `:` as "inverse cons" for `x : (y : xs)`, which means that `x` is th`enter code here`e head, and `y` is the next item after the head.
- can also use the constant `[]` in place of the tail (this only matches patterns where the tail is *exactly* the empty list)

```
-- favs.hs
favorites :: [String] -> String
favorites [] = "You have no favorites."
favorites (x:[]) = "Your favorite is " ++ x
favorites (x:y:[]) = "You have two favorites: "
 ++ x ++ " and " ++ y
favorites ("chocolate":xs) =
  "You have many favs, but chocolate is #1!"
favorites (x:y:_) =
   "You have at least three favorites!"
```

- if you use `[]` instead, only matches lists of an exact length; each item in the list is just an item (there are no tails, etc.).

```
-- lists.hs\
whats_in_your_list [10] =
    "Your list has just 1 value which is 10."
whats_in_your_list [a,b] = "Your list has 2 values: "
    ++ (show a) ++ " and " ++ (show b)
whats_in_your_list [9,_,c] =
    "Starts with 9 and its third item is:"  ++ (show c)
whats_in_your_list _ =
    "Your list had something else in it."
```

First-class and higher-order functions

A language with **first-class functions** treats functions like any other data. They can be:

- stored in variables or data structures
- passed as arguments to other functions
- returned as values by functions

Enables more *efficient program decomposition* and allow functions to *generate new functions from scratch*

higher-order function either:

- a function that accepts another function as an argument, and/or
- a function that returns another function as its return value

```
insult :: String -> String
insult name = name ++ " is so cringe!"

praise :: String -> String
praise name = name ++ " is dank!"

talk_to :: String -> (String -> String) -> String
```

```
talk_to name talk_func
  | name == "Carey" = "No comment."
  | otherwise = (talk_func name)
```

- `talk_to` is a higher-order function: its second argument, `talk_func`, is itself a function that takes in a `String` and returns a `String`
  - reflected in its type: `(String -> String)`;
- note the parentheses in the type declaration to denote a function-type parameter.

```
get_pickup_func :: Int -> (String -> String)
get_pickup_func born
  | born >= 1997 && born <= 2012 = pickup_genz
  | otherwise = pickup_other
 where
   pickup_genz name = name ++ ", you've got steez!"
   pickup_other name = name ++ ", you've got style!"
```

The higher-order function `get_pickup_func` returns different functions according to the argument `born`. The returned function can be stored into variable and invoked later.

```
ghci> pickup_fn = get_pickup_func 2003
ghci> pickup_fn "Jayathi"
"Jayathi, you've got steez!"
ghci> get_pickup_func 1971 "Carey"
"Carey, you've got style!"
```

first-class and higher-order functions are a pillar of modern programming languages

- providing comparison functions for sorting
- as callbacks when events trigger
- enabling multi-threading

Map, Filter, Reduce

- set of higher-order utility functions to process lists
- A **mapper** function performs a one-to-one transformation from one list of values to another list of values using a *transform function*
- A **filter** function filters out items from one list of values using a *predicate function* to produce another list of values
- A **reducer** function operates on a list of values and collapses them into a single output value
- typically use one or more mappers and filters followed by reducer(s) to crunch data, e.g.: `result = reduce(map(filter(input)))`.

Map

- maps a list of values to another list of values of the same length
- Haskell `map` accepts two parameters:
  1. A function to apply to every element of a list
  2. A list to operate on
- type signature: `map :: (a -> b) -> [a] -> [b]`

    - first argument is a function that maps an individual item from a value of type `a` to a value of type `b` (`a` and `b` can be the same)
    - The second argument is a list of type `a`
    - The return value is a list of type `b`

```haskell
is_even :: Int -> Bool
is_even x = x `mod` 2 =0
```

```
ghci> map is_even [1,2,3,4,6]
[False,True,False,True,True]
ghci> map reverse ["mouse","cat","fly"]
["esuom","tac","ylf"]
```

Implementation:

```haskell
-- map.hs
map :: (a -> b) -> [a] -> [b]
map func [] = []
map func (x:xs) =
  (func x) : map func xs
```

Filter

- filters items from an input list to produce a new output list.
- Haskell `filter` accepts two parameters:
    1. A function that determines if an item in the input list should be included in the output list
    2. A list to operate on
- type signature: `filter :: (a -> Bool) -> [a] -> [a]`
    - The first argument is a predicate function that determines if a value from the input list should be included in the output
    - The second argument is a list of type `a`
    - The returned list include all the items that passed the filter

Implementation:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter predicate [] = []
filter predicate (x:xs)
 | (predicate x) = x : (filter predicate xs)
 | otherwise = filter predicate xs
```

Reducers (foldl/foldr)

- a function that combines the values in an input list to produce a single output value (eg: reduce all the items from a list into a sum or product)
- Each reducer takes three inputs:
    1. A function that processes each of the elements
    2. An initial "accumulator" value
    3. A list of items to operate on

Foldl

```
# not functional yet
foldl(f, initial_accum, lst):
  accum = initial_accum
  for each x in lst:
    accum = f(accum, x)
  return accum
```

The core logic of `foldl` all happens around the accumulator variable `accum`:

- first, set accume to `initial_accum`, the second argument to the function (often be 0, 1 or empty list)
- Next, `accum` is updated as we loop through each item. Each time the function `f` is used to "accumulate" the item to `accum`.
- final value of `accum` is returned

Foldr

```
foldr(f, initial_accum, lst):
  accum = initial_accum
  for each x in lst from back to front:
    accum = f(x, accum)
  return accum
```

There are two differences between `foldr` and `foldl`:

- The iteration order of list is reversed in `foldr`
- The order of arguments of `f` are different

Haskell code for `foldl` and `foldr`:

```
foldl f accum [] = accum
foldl f accum (x:xs) =
  foldl f new_accum xs
 where new_accum = (f accum x)

foldr f accum [] = accum
foldr f accum (x:xs) =
  f x (foldr f accum xs)
```

each new value x is "folded" into the accumulator as it's processed.

- `foldl` is left-associative: `f( ... f(f(accum,x1),x2), ..., xn)`
- `foldr` is right-associative: `f(x1, f(x2, ... f(xn, accum) ... ))`

under some circumstances, `foldr` can process infinite lists!

Lambda Functions

- A function with no name
- Note: lambdas are another manifestation that functions are the first-class citizens
  - Allows functions to be written as values without names, just like values of other types (integers, strings, etc.)
- can be regarded as more fundamental than named functions: defining named function like `a_func x y = x ++ y` can be regarded as syntatic sugar of `a_func`

`= \x y -> x ++ y`.

Syntax:

```
\param_1 ... param_n -> expression
```

can't just define a stand-alone lambda, save the lambda function we created in a variable

```
ghci> a_func = \x y -> x++y
ghci> a_func [1,2,3] [4,5]
[1,2,3,4,5]
```

Can use a lambda to generate new functions

```
wrapFuncWithAbs func = (\x -> abs (func x))
twox = 2*x

abs2x = wrapFuncWithAbs twox
```

Closures

```
slopeIntercept m b = (\x -> m*x + b)

twoxPlusOne = slopeIntercept 2 1
```

created a function generator `slopeIntercept`:

- accepts two parameters: `m` and `b`
- returns a new function a new function that takes argument `x` and computes `y = m*x + b`

Then we create a function `twoxPlusOne` by calling `slopeIntercept` with `m` equals to 2 and `b` equals to 1. In the process, Haskell "captures" the specific values of `m` and `b` along with the expression `\x -> m*x + b`. This way, when we next call `twoxPlusOne`. Haskell will use `m=2` and `b=1`.

```
ghci> twoxPlusOne 9
19
```

- **closure** - the combination of a *lambda expression* with a snapshot of all *"captured" values* (like `m` and `b`)
- "captured" or **free variable** is any variable that is *not an explicit parameter* to the lambda
  - in `\x -> m*x + b`, `m` and `b` are free variable, and `x` is not (as it appears in the parameter)
- **bound variables** are those that appear in the lambda parameters

Formally, a closure is a combination of:

> 1. *A function of zero or more arguments that we wish to run at some point in the future*
> 2. *A list of "free" variables and their values that were captured at the time the closure was created*

When a closure later runs, it uses the values of the free variables captured at the time it was created.

- free variables can't be stored on stack, would lose them after return
- must be stored on heap

Currying

currying transforms a function of multiple arguments to a series of functions of a single argument.

```
function f(x, y, z) {
  return x + y + z;
}

// becomes
function f(x) {
  function g(y) {
    function h(z) {
      return x + y + z;
    }
    return h;
  }
  return g;
}
```

original function is converted into a series of nested functions:

- The number of nested level equals to the number of argument of the original function
- Each function takes a single argument in sequence, starting from the outmost one
- The outer functions returns the nested function in the next level
- The inner-most function does the original computation

Say we wanted to call our original function like this: `f(10, 20, 30)`. We can curry it:

```
temp_func1 = f(10);
temp_func2 = temp_func1(20);
final_result = temp_func2(30);
```

Each call above fills in one variable (`x` and `y` respectively). When we do the final call, only the last parameter is needed and we get the result we want.

Or more concisely, we can do it on a single line: `final_result = f(10)(20)(30)`

Every function with two or more parameters can be represented in curried form!

Formally, Currying is the concept that you can represent any function that takes multiple arguments by another that takes a single argument and returns a new function that takes the next argument, etc.

- `y = f(a, b, c)` -> `y = ((f_c a) b) c` .

Each function takes one argument and returns another function as its result (except for the last function which returns the result).

Pseudocode for currying:

```
Curry(Function f)
  e = The expression/body of function f
  For each parameter p from right to left:
    f_temp = Define a new lambda function with:
      1. p as its only parameter
      2. e as its (expression)
    e = f_temp
return f_temp
```

Let's use the procedure above to curry the function `mult3 x y z = x*y*z`

- At first, `e` is equal to `x*y*z`
- Then we enter the for loop
  - In the first iteration, `z` is the parameter to be handled. `f_temp = \z -> x*y*z`
  - Next, we handle `y` and `f_temp = \y -> (\z -> x*y*z)`
  - Finally, we handle `y` and we get `f_temp = \x -> (\y -> (\z -> x*y*z))`

Thus, the curried version `mult3c = \x -> (\y -> (\z -> x*y*z))`.

Let's see how we call our curried function with `x=2`, `y=3` and `z=5`!

```
ghci> mult3c = \x -> (\y -> (\z -> x*y*z))
ghci> f1 = mult3c 2
ghci> f2 = f1 3
ghci> result = f2 5
30
```

- `f1` is obtained by calling the lambda of `mult3c` with `x=2`. It is: `\y -> (\z -> 2*y*z)`.
- `f2 = \z -> (2*3*z)`
- Note: When a inner lambda has the same parameter name as the outer one. The inner one takes precedence and *shadows* the outer one. The shadowed parameter will not get replaced
  - ex: `(\x -> ((\x -> x) (x + 1))) 3` is `((\x -> x) (3 + 1))`

Every time you define a function of more than parameter in Haskell, Haskell automatically and invisibly **curries it** for you.

- `(((mult3 2) 3) 5)` is valid
- our usual way of calling functions `mult3 2 3 5` looks like a *single call* with three arguments, but it's really *three calls* to three different functions! Exactly identical to `(((mult3 2) 3) 5)`

Type Signatures

```
mult3 :: Double -> Double -> Double -> Double
mult3 x y z = x * y * z
```

- should really look like this

```
mult3 :: Double -> (Double -> (Double -> Double))
mult3 x y z = x * y * z
```

- latter form is more *fundamental*,
- use more syntactic sugar to remove the parentheses. (We can regard the `->` "operator" in type signatures as **right-associative**)

In Type Signatures: a -> (b -> c) -> d = a -> ((b -> c) -> d)

Why curry functions?

- enables partial function application
- But mostly it appears to be motivated by theorists! can facilitate program analysis and is consistent with the *lambda calculus*, the mathematical foundation of functional programming.

Partial Function Application

- calling a function with less arguments than it takes

Formally: an operation where we define a new function `g` by combining:

- An existing function `f` that takes one or more arguments, with
- Default values for one or more of those arguments

The new function `g` is a specialization of `f`, with hard-coded values for some of `f`'s parameters.

Once defined, we can then call `g` with those arguments that have not yet been hard-coded.

Partial function application is deeply related with currying we have just learned. In fact, when we used `((mult3 2) 3) 5` earlier, we are implicitly doing partial function application

```
ghci> mult3 x y z = x*y*z
ghci> partial = mult3 2
ghci> partial 3 5
30
```

Partial function applications can be useful when combined with pre-existing functions and operators

```
ghci> add5 = (+) 5
ghci> add5 100
105
```

All of Haskell's operators are just functions! So the `+` sign is basically a function of two arguments.

```
ghci> map (/ 10) [100,200,300]
[10.0,20.0,30.0]
ghci> map (10 /) [100,200,300]
[0.1,5.0e-2,3.333333333333333e-2]
```

- can apply partial application to *either operands of infix operators*
- The function produced by the first line is equivalent to `\x -> x / 10`
- the second line produces `\x -> 10 / x`

Here are more examples:

```
ghci> filter (>= 6) [2,4,6,8,10,3]
[6,8,10]
ghci> map (++ " is LIT!") ["CS32","CS131"]
["CS32 is LIT!","CS131 is LIT!"]
ghci> filter (`elem` ['A'..'Z']) "Not Every Resistor Drives current"
"NERD"
```

We can even use partial function application to define a partially-specified mapper function:

```
ghci> cuber = map (\x -> x^3)
ghci> cuber [2,3,5]
[8,27,125]
```

Algebraic Data Types

- ADTs are user-defined data type that can have multiple fields.
- The closest thing to algebraic data types would be C++ structs and enumerated types
- The simplest algebraic data types are just like C++ enums:

```
data Drink = Water | Coke | Sprite | Redbull
data Veggie = Broccoli | Lettuce | Tomato
data Protein = Eggs | Beef | Chicken | Beans
```

- The pipe (|) character means "OR"
- Each choice for a given type is called a **variant** or a "kind"

equivalent in C++:

```
enum Protein { Eggs, beef, chicken, Beans };
```

We can also define more complex ADTs where each variant also has one or more fields (like C++ struct).

```
data Meal =
 Breakfast Drink Protein |
 Lunch Drink Protein Veggie |
 Fasting
```

The definition of first variant `Breakfast` means that it is comprised of `Drink` and `Protein` fields. Equivalent in C++:

```
struct Breakfast {
  Drink d;
  Protein p;
}
```

NOTE: fields in Haskell does not have a name, they are distuished by the order they appear in the definition

Note that the variants with and without fields can coexist within a single ADT, the `Fasting` above is just an example.

Here are some examples to define variables with ADT:

```
careys_meal = Breakfast Redbull Eggs
pauls_meal = Lunch Water Chicken Broccoli
davids_meal = Fasting
```

Syntax

```
data Color = Red | Green | Blue

data Shape =
  Circle
   Float Float -- x, y
   Float       -- radius
   Color |
  Rectangle
   Float Float -- x1, y1
   Float Float -- x2, y2
   Color
```

- To define a new ADT, we use the `data` keyword
- The syntax can be represented as `data TypeName = Variant1 | Variant2 | ... | VariantN`
- The initial of type names should be *capitalized* (to differentiate with type variables)
- The initial of variant names should be capitalized as well.
- The general syntax for defining a variant that has fields is: `VariantName Type1 Type2 ... TypeN`

Each time you define a variant with fields, Haskell implicitly creats a "constructor". In the code above a `Circle` constructor is created with four parameters (three Floats and a Color) to create a new `Circle`.

Add `deriving Show` at the end of a definition to enable us to print out variable values in GHCi

```
ghci> c = Circle 5 6 10 Red
ghci> c
Circle 5.0 6.0 10.0 Red
ghci> r = Rectangle 0 0 5 6 Blue
ghci> r
Rectangle 0.0 0.0 5.0 6.0 Blue
ghci> :t r
r :: Shape
ghci> colors = [Red, Red, Blue]
ghci> colors
[Red,Red,Blue]
```

Pattern Matching with ADT

```
data Shape =
   Shapeless
 | Circle
```

```
    -- x      y     rad
    Double Double Double
  | Rectangle
    -- x1    y1     x2     y2
    Double Double Double Double
  deriving Show

getArea :: Shape -> Double
getArea Shapeless = 0
getArea (Circle _ _ r) = pi * r^2
getArea (Rectangle x1 y1 x2 y2) =
  (abs (x2-x1)) * (abs (y2-y1))
```

We focus on the `getArea` function:

- The first line is the type signature. As this functions operates on various types of `Shape`
- Then follows the actual pattern matching
  - The first line means: "If the user passes in a Shapeless variable, then return zero".
  - The second line handles the `Circle` case. It gets the third field (radius), placing it into variable `r` and computes the area.
  - The last line handles `Rectangle`

NOTE: can combine pattern matching on ADT with lists and tuples to create complex patterns. For example, the following function counts the occurrences of `Circle` within a list of `Shape`

```
count_circle :: [Shape] -> Integer
count_circle [] = 0
count_circle ((Circle _ _ _):xs) = 1 + count_circle xs
count_circle (_:xs) = count_circle xs
```

Creating Trees with ADT

```
-- Binary Search Tree
data Tree =
  Nil |
  Node String Tree Tree
```

A tree data type has two possible variants:

- An empty variant indicating that there is no node, like a `nullptr`
- A node variant representing actual nodes.

In this definition, each Node holds a String value, and two other tree data items that can be either `Nil` or `Node`

```
ghci> empty_tree = Nil
ghci> one_node = Node "Lit" Nil Nil
ghci> left_child = Node "Boomer" Nil Nil
ghci> right_child = Node "Zoomer" Nil Nil
ghci> root = Node "Cheugy" left_child right_child
```

```
ghci> root
Node "Cheugy" (Node "Boomer" Nil Nil) (Node "Zoomer" Nil Nil)
```

Search function:

```
search Nil val = False
search (Node curval left right) val
 | val == curval = True
 | val < curval = search left val
 | otherwise = search right val
```

Add node:

- not so easy because values are immutable
- suppose we create a node N and we want to make it the child of existing node M
- can't just find a node M and change its left or right child to point to the new N!
- must create a new node M', let it point to N and replace the node M
- but now the new node M' is not part of the tree either; repeat for M's parent
- Eventually we can do this all the way to the root

```
insert Nil val =
  Node val Nil Nil

insert (Node curval left right) val
  | val == curval =
      Node curval left right
  | val < curval =
      Node curval (insert left val) right
  | val > curval =
      Node curval left (insert right val)
```

- note, if we find a node that have value `val` we must create an exact copy of it to return because there's no other way to refer to it
- sounds wasteful, however we only need to generate replacement nodes for nodes between our new node and the root.
  - if a tree is balanced only need to create `log2(n)` new nodes
- old nodes **garbage collected** - a language feature that automatically reclaims unused variables
  - all functional languages have built-in GC
  - doesn't impact Big-O
- Some data structures perform poorly if immutable:
  - immutable hash tables: must regenerate the full array of n buckets every time you add a new item

Note: For a generic ADT, need to specify generic type

- Here, need the `a` on the LHS to specify generic tree:

```
data Tree = Empty | Node Integer [Tree]
-- to make into a generic tree
data GenericTree a = Empty | Node a [GenericTree]
```

Influencer Alert: Immutability

Many languages now provide immutable data structures – it helps simplify code, reduce bugs, and ease multi-processing.

Examples include Google's Guava library for Java, and immutable.js for JavaScript.